National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Memory Allocation

IOC5226 Operating System Capstone

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao Tung University

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - MIT 6.828 Operating system engineering class, 2018
  - MIT 6.004 Operating system, 2018
  - Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC

Outline

- Dynamic memory allocation
- Buddy memory allocator
- Slab memory allocator

Dynamic memory allocation

- How does the OS manage memory of a single process ?
  - Each process has contiguous logical address space
- **Static (compile-time) allocation** is not always a good choice
  - Recursive procedures
    - Data dependencies are hard to predict
  - Complex data structures
    - Link-list, tree, graph (ptr = malloc(x); free(ptr))
- **Dynamic allocation**
  - Stack allocation
  - Heap allocation

Stack organization

- Stack grows happens via
  - **mremap ()** : remap a virtual memory address
- When is it useful ?
  - Memory allocation and freeing are partially predictable
  - Examples
    - Procedure call frames, tree traversal, recursion
- Advantages
  - Keeps all the free space contiguous
  - Simple and efficient to implement
- Disadvantages
  - Not appropriate for all data structures

alloc(A)
alloc(B)
alloc(C)
free(C)
free(B)
free(A)

Heap organization

- Allocate from random locations
  - Memory contains **allocated** areas and **free** areas
- When is it useful ?
  - Allocation and release are unpredictable
  - Arbitrary list structures, complex data organizations
    - E.g. new in C++, malloc() in C
- Advantage: works on arbitrary allocation and free patterns
- Disadvantage: End up with small chunks of free space

Stack vs heap allocation

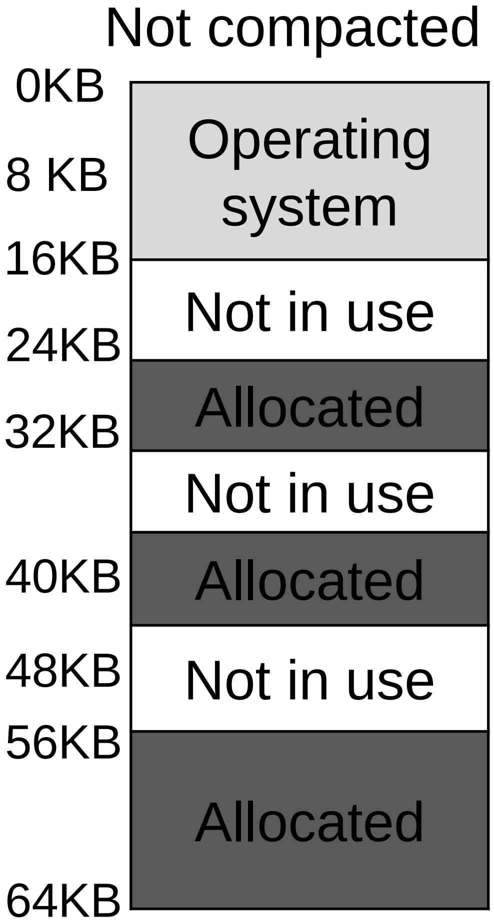
Parameter	Stack	Heap
Basic	Allocated in a contiguous block	Allocated in a random order
Allocation	Automatic by compiler	Manual by programmer
Main issue	Storage of memory	Memory fragmentation
Safety	Thread safe, data only accessed by owner	Not thread-safe, data stored visible to all threads
Flexibility	Fixed-size	Resizing is possible
Access time	Fast	Slow

Fragmentation

- **Internal fragmentation**
  - Waste space when you round an allocation up
- **External fragmentation**
  - When you end up with small chunks of free memory that are too small to be useful

External fragmentation

- **External fragmentation**
  - Full of little holes of free space
  - Have a number of segments per process
  - Each segment might be a different size
  - It is difficult to allocate new segments
- **Compact physical memory**
  - Rearranging the existing segments
  - Compaction is expansive
  - Best-fit, worst-fit, first-fit, buddy algorithm



External fragmentation (cont.)

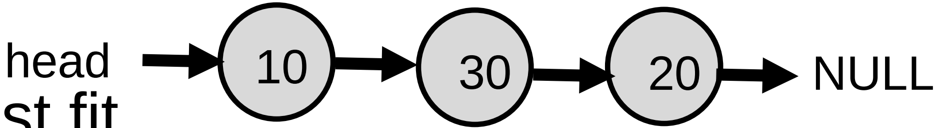
- **When does external fragmentation occur ?**
  - The free space consists of variable-sized units
  - This arises in a user-level memory allocation library (malloc())
  - Chops segments into little pieces of different sizes
- **Problems of the external fragmentation**
  - No single contiguous space that can satisfy the request
  - Even the total amount of free space exceeds the size of requests
  - E.g. A request 15 bytes will fail even though there are 20 bytes free

Memory allocation strategies

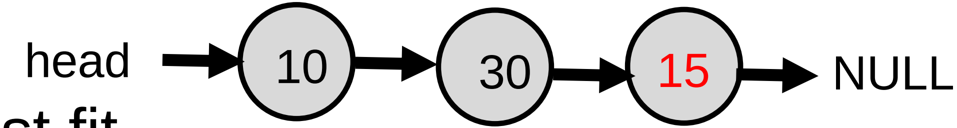
- **Best fit**
  - Return a block that is close to what the user asks
  - Try to reduce wasted space
  - Perform an exhaustive search for the correct free block penalty
- **First fit**
  - Find the first block that is big enough and returns the requested amount to the user
  - Has the advantage of speed – no exhaustive search
  - How the allocator manages the free list's order becomes an issue ?

Case study: memory block fitting

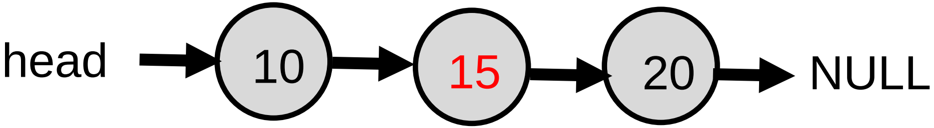
- Envision a free list with three elements on it
  - Assume an allocation request of size 15



- Best fit
  - Search the entire list and find that 20 was the best fit



- First fit
  - Find the first free block that can satisfy the request



Designing memory allocator issues

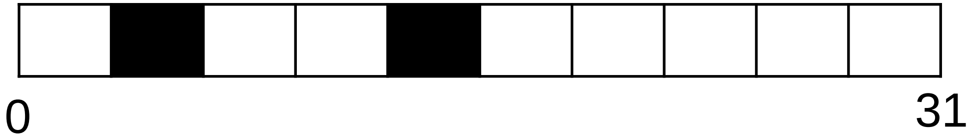
- How to keep track of the size of a block ?
- How to keep track of which blocks are in use and free ?
- How to align memory space if a block is smaller than the free block we find ?
- How to pick a block to use for allocation ?
- How do re-insert freed block ?

Buddy allocator

- Fast, simple allocation for blocks that are 2^n bytes
- Allocation restrictions
  - Block sizes: 2^n
- Allocation strategy for k bytes
  - Raise allocation request to the nearest 2^n
  - Search free list for appropriate size
    - Recursively divide large blocks until reach block of correct size
  - Free strategy
    - Recursively coalesce block with buddy if buddy free
    - May coalesce lazily to avoid overhead

Buddy allocator issues

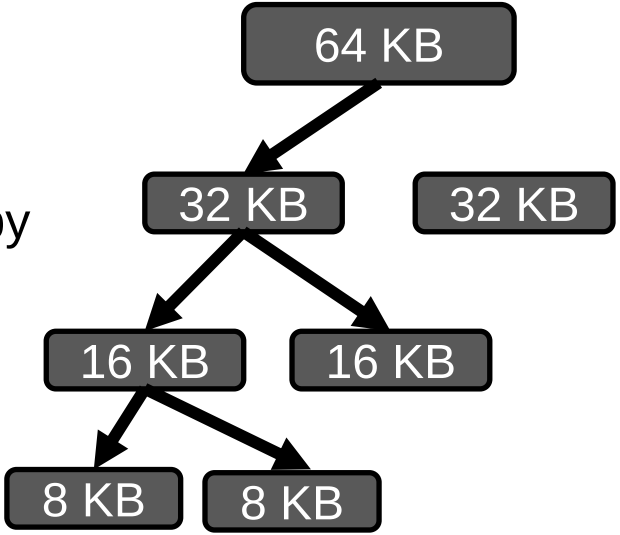
- **Memory fragmentation**
  - Buddy allocator still leads to few reserved pages that prevent the allocation of larger contiguous blocks
- **Performance**
  - Very fast, since the simple binary shift or bit change arithmetic



Memory layout of the buddy allocator

Buddy allocation

- **Binary buddy allocator**
  - Free memory as one big space of size 2^N
  - Recursive search by dividing free space by two until a block that is big enough to accommodate the request is found
  - **Internal fragmentation** as only allowed to power-of-two-sized block
  - Check whether the “buddy” 8KB is free when returning the 8KB block to the free list
  - Keep coalescing when the buddy is still free
  - Making coalescing simple



Case study: buddy allocation

Memory blocks

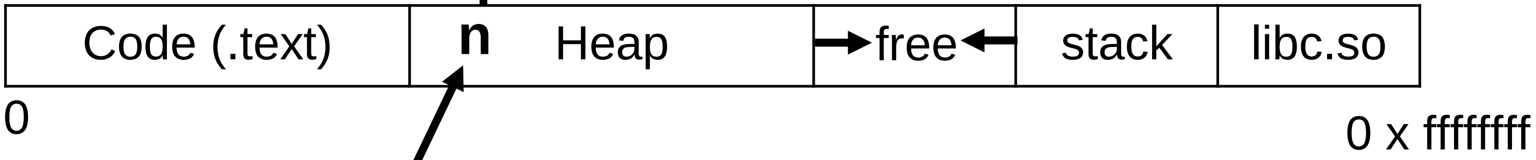
0	1	2	3	4	5	6	7

- In a memory
  - Block 0, 4, 5, 6, 7 is used
  - Will buddy allocator merges block 1 and 2 if both of them are free ?
    - No !! Block 1 and 2 are not buddy

```
static inline unsigned long _find_buddy_pfn
                              (unsigned long page_pfn, unsigned int order)
{
          return page_pfn ^ (1 << order);
}
```

How to allocate memory ?

Virtual address space




```
int main () {
        struct foo *x = malloc(sizeof(struct foo));
        ….
}
```

```
void* malloc (ssize_t n) {
        if(heap empty)
                mmap(); // add pages to heap and find a free block of size n
}
```

malloc() issues

- How to implement malloc() or new ?
  - Calls **sbrk()** to request more contiguous memory from OS
  - Add small header to each block of memory
    - Pointer to next free block

sbrk() increments the program's data space by increment bytes. Calling sbrk() with an increment of 0 can be used to find the current location of the program break.

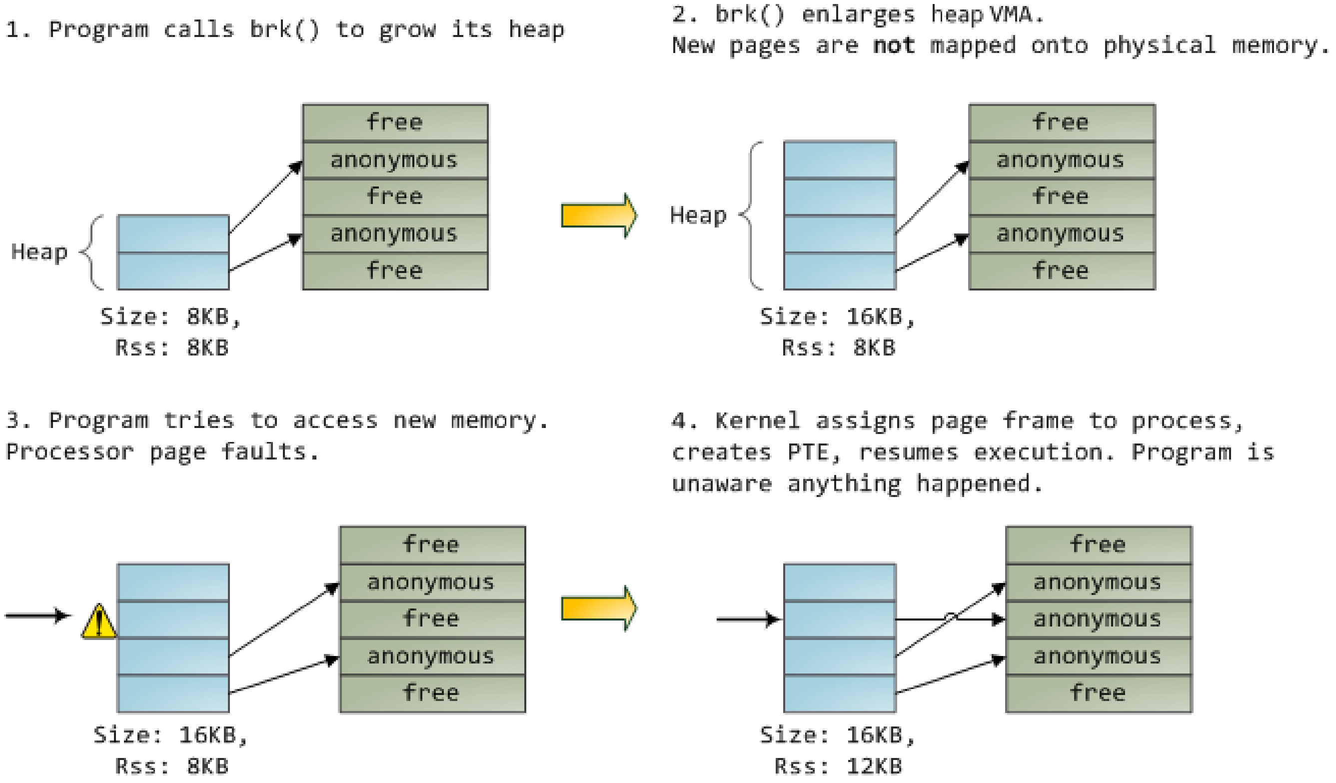
On success, sbrk() returns the previous program break. (If the break was increased, then this value is a pointer to the start of the newly allocated memory). On error, (void *) -1 is returned, and errno is set to ENOMEM.

https://pages.cs.wisc.edu/~eli/537/lectures/lecture12.2x2.pdf

Enlarge VMA

brk() sets the end of the data segment to the value specified by addr, when that value is reasonable, the system has enough memory, and the process does not exceed its maximum data size.

On success, brk() returns zero. On error, -1 is returned, and errno is set to ENOMEM.



Source: http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/

https://pages.cs.wisc.edu/~eli/537/lectures/lecture12.2x2.pdf

Reclaiming free memory

- When can dynamically-allocated memory be freed ?
  - Explicitly call free()
  - Hard, can't be recycled until all sharers are finished
    - Sharing is indicated by the presence of pointers to the data
- Two possible problems
  - **Dangling pointers**
    - Recycle storage while it's still being used
  - **Memory leaks**
    - Forget to free storage even when can't be used again
    - Not a problem for short-lived user processes
    - Issue for operating systems and long-running applications

Garbage collection

- **Idea**
  - No free() operation
  - Storage freed implicitly when no longer referenced
- **Approach**
  - When system needs storage, examine and collect free memory
- **Advantages**
  - Makes life easier on the application programmer

https://pages.cs.wisc.edu/~eli/537/lectures/lecture12.2x2.pdf

Mark and sweep

- Requirements
  - Must be able to find all objects
  - Must be able to find all pointers to objects
  - Compiler must cooperate by marking type of data in memory
- **Two passes**
  - **Pass 1: Mark**
    - Start with all statically-allocated and procedure-local variables (on stack)
    - Mark each object
    - Recursively mark all objects can reach with a pointer
  - **Pass 2: Sweep**
    - Go through all objects, free those that aren’t marked

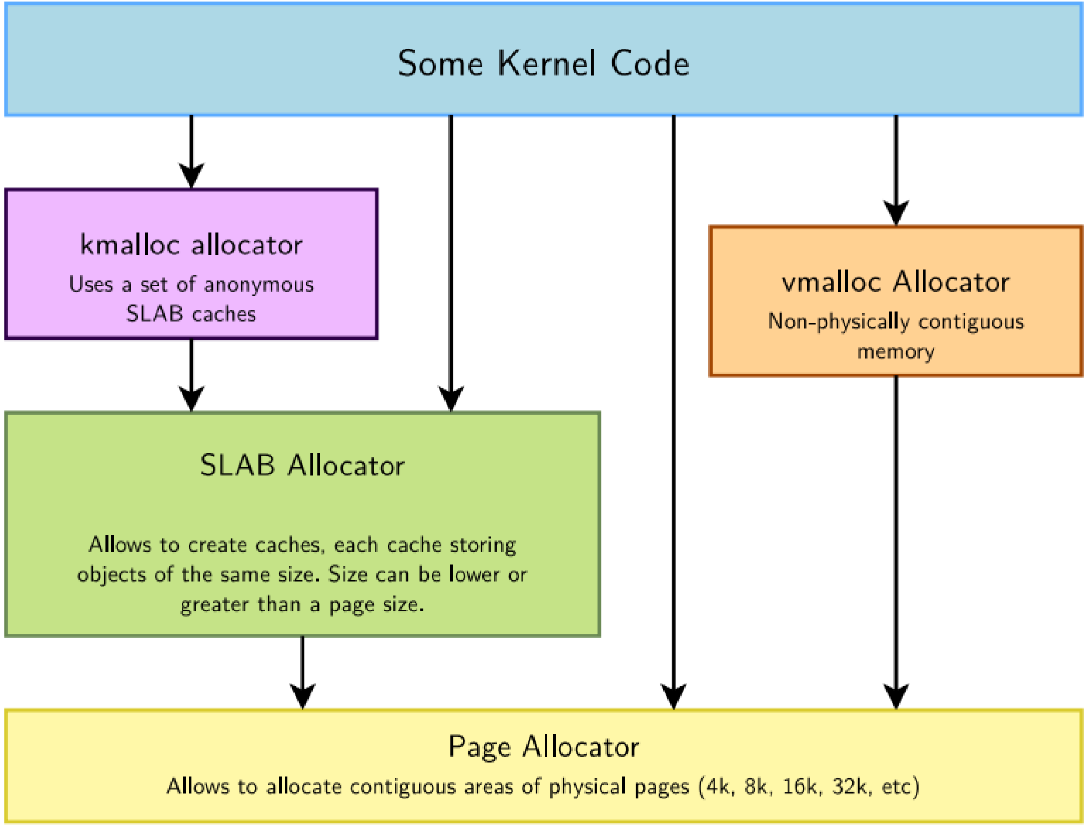
https://pages.cs.wisc.edu/~eli/537/lectures/lecture12.2x2.pdf

Garbage collection in practice

- **Disadvantages**
  - **Expansive**: 20% or more of CPU
  - **Difficult to implement**
    - Execute program during garbage collection (incremental)
- Languages with garbage collection
  - LISP
  - Java

https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

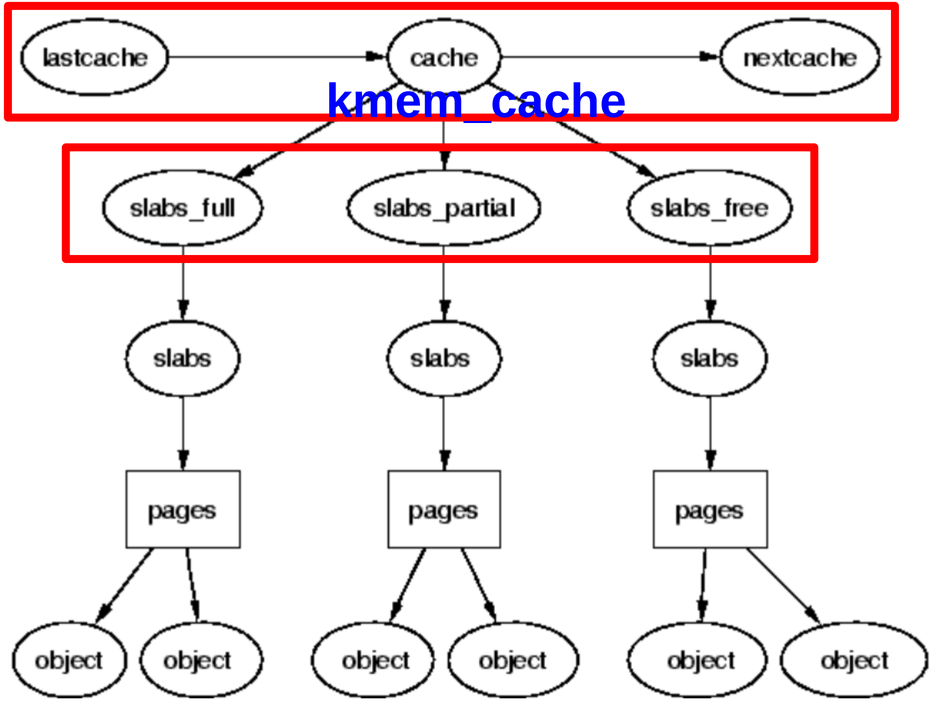
Linux kernel allocators



Page allocator

- Appropriate for medium-size allocations
- A page is usually **4KB** that is dependent to the hardware
- **Buddy allocator strategy**
  - Only allocations of power of two numbers of pages such as 1, 2, 4, 8, 16 pages, etc.
  - Typical maximum size is 8192 KB
  - The allocated area is contiguous in the kernel virtual address space
  - Maps to physically contiguous pages

What is slab ?



https://www.kernel.org/doc/gorman/html/understand/understand011.html

- **Slab**
  - a chunk of contiguous pages
  - A container of objects
  - Allocates a number of objects to the slabs associated with that cache
- **Cache chain**
  - A variable number of caches linked on a doubly linked circular list
  - Kmem_cache_s manages objects such as mm_struct or fs_cache

What is slab ? (cont.)

- The slab allocator manages the objects in a cache
  - A slab contains one or more pages, divided into equal-sized objects
  - When cached created, allocate a slab, divided the slab into free objects
  - If a slab is full of used objects, next object comes from an empty/new slab
- **Benefits**
  - Fast memory allocation
  - Some of the object fields may be reusable; no need to initialize again

Motivation of the slab allocator

- **The kernel needs**
  - Many different **temporary objects**
  - Such as the mm_struct, inode, files_struct structures
- **Temporary kernel objects**
  - Very small and very large size
  - They are often allocated and freed
  - Require to perform object allocation efficiently
- **Drawbacks of the buddy allocator**
  - Its free areas are composed of entire frames of memory (too large for various object size)
  - Align objects with power of two size has a negative impact on the use of the process cache

Principle of the slab allocator

- The allocation of **small memory blocks**
  - Eliminate internal fragmentation caused by a binary buddy allocator
  - Two caches of small memory buffers (32 – 131072 bytes)
  - kmalloc() is provided for allocate objects in these small cache buffers
- The **caching** of commonly used objects
  - The system doesn’t waste time allocating, initializing and destroying objects
- **The better utilization of hardware cache**
  - aligning objects to the L1 or L2 caches

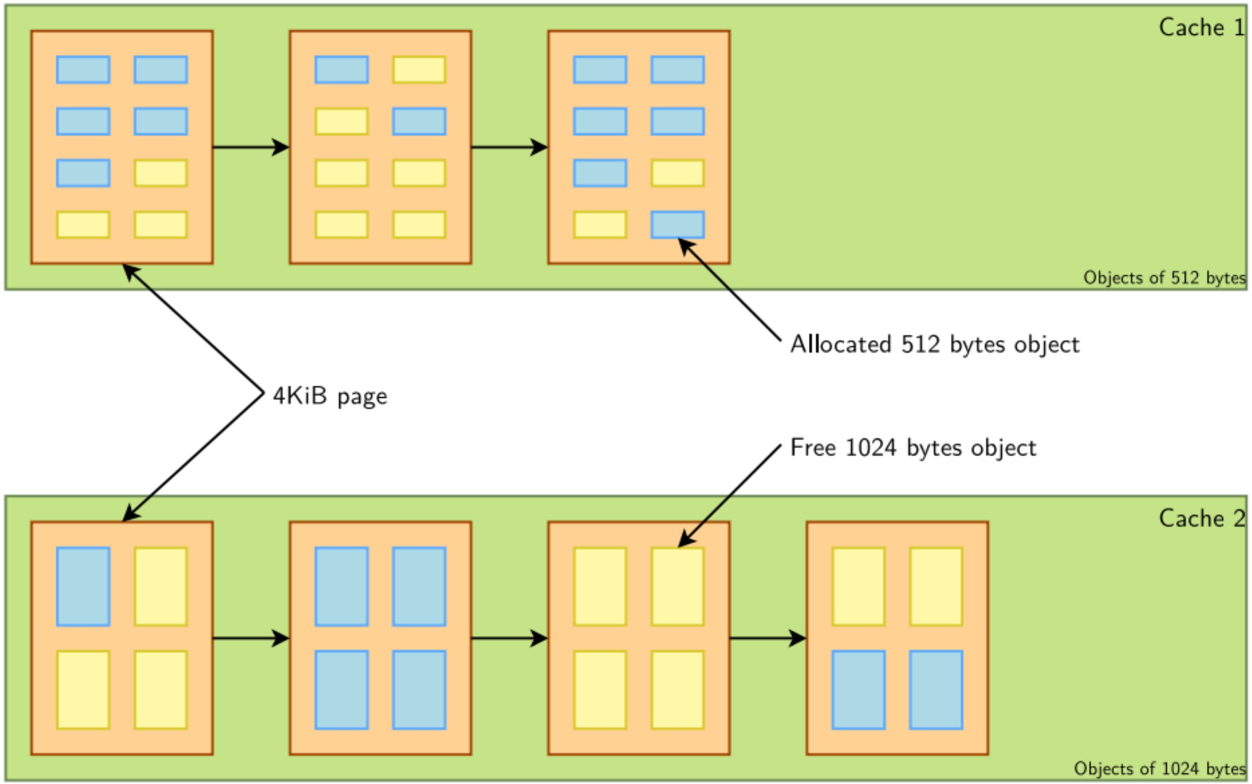
The slab allocator

- **The slab allocator**
  - The default cache allocator (at least as of early Linux kernel 2.6, Solaris)
  - A given cache allocates a specific type of object
    - E.g. a cache for file descriptors, a cache for inodes
  - **Motivation**
    - The kernel often **spends much of its time on allocating, initializing and freeing the same object**
    - Reduce the number of references to the buddy allocator
  - **Basic idea**
    - Have **caches** of commonly used objects kept in an initialized state for use by the kernel

The slab allocator (cont.)

- The SLAB allocator
  - Allow to **create caches**, which contain a set of objects of the same size
  - The object size can be smaller or greater than the page size
  - Takes care of growing or reducing the size of cache as needed
  - Uses the page allocator to allocate and free pages
  - SLAB caches are used for data structures that are present in kernel instances
    - Directory entries, file objects, network packet descriptors etc..
    - See /proc/slabinfo

The slab allocator(cont.)



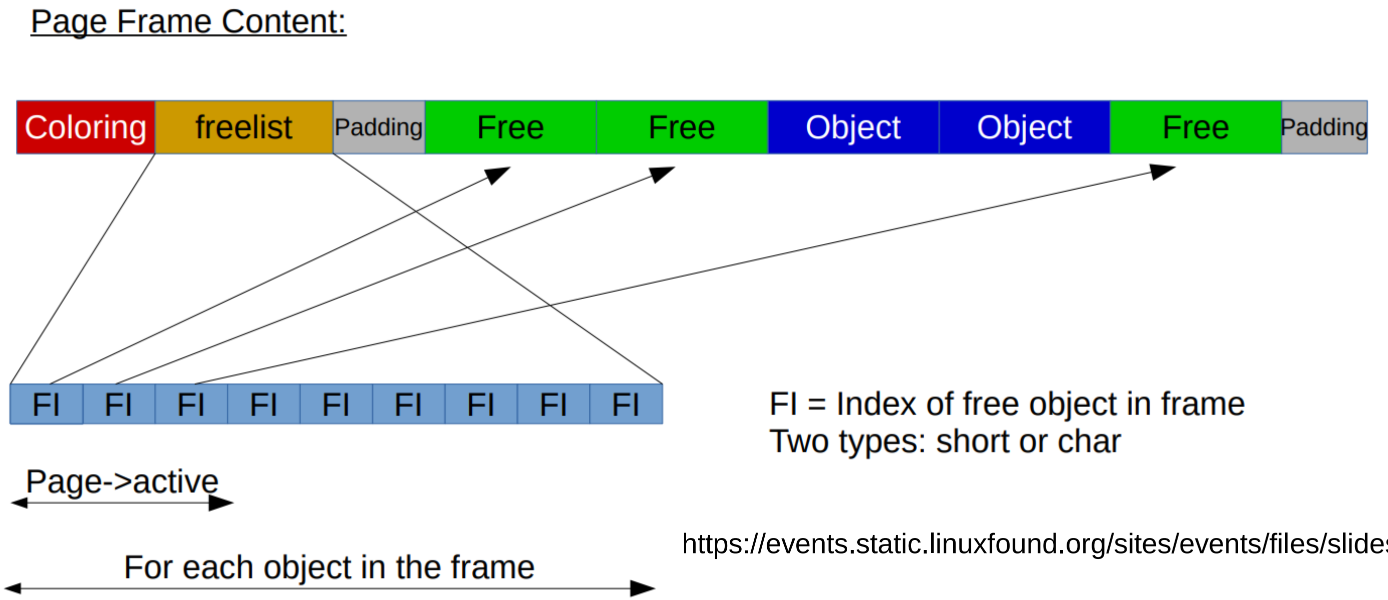
https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

Alternative slab allocators

- **SLOB allocator**
  - Designed for small systems
  - As compact as possible
- **SLAB allocator**
  - As cache friendly as possible
- **SLUB allocator**
  - Designed for large systems
  - Minimize memory overhead
  - Execution time friendly

SLAB per frame freelist management

- Multiple requests for free objects can be satisfied from the same cache line without touching the object contents

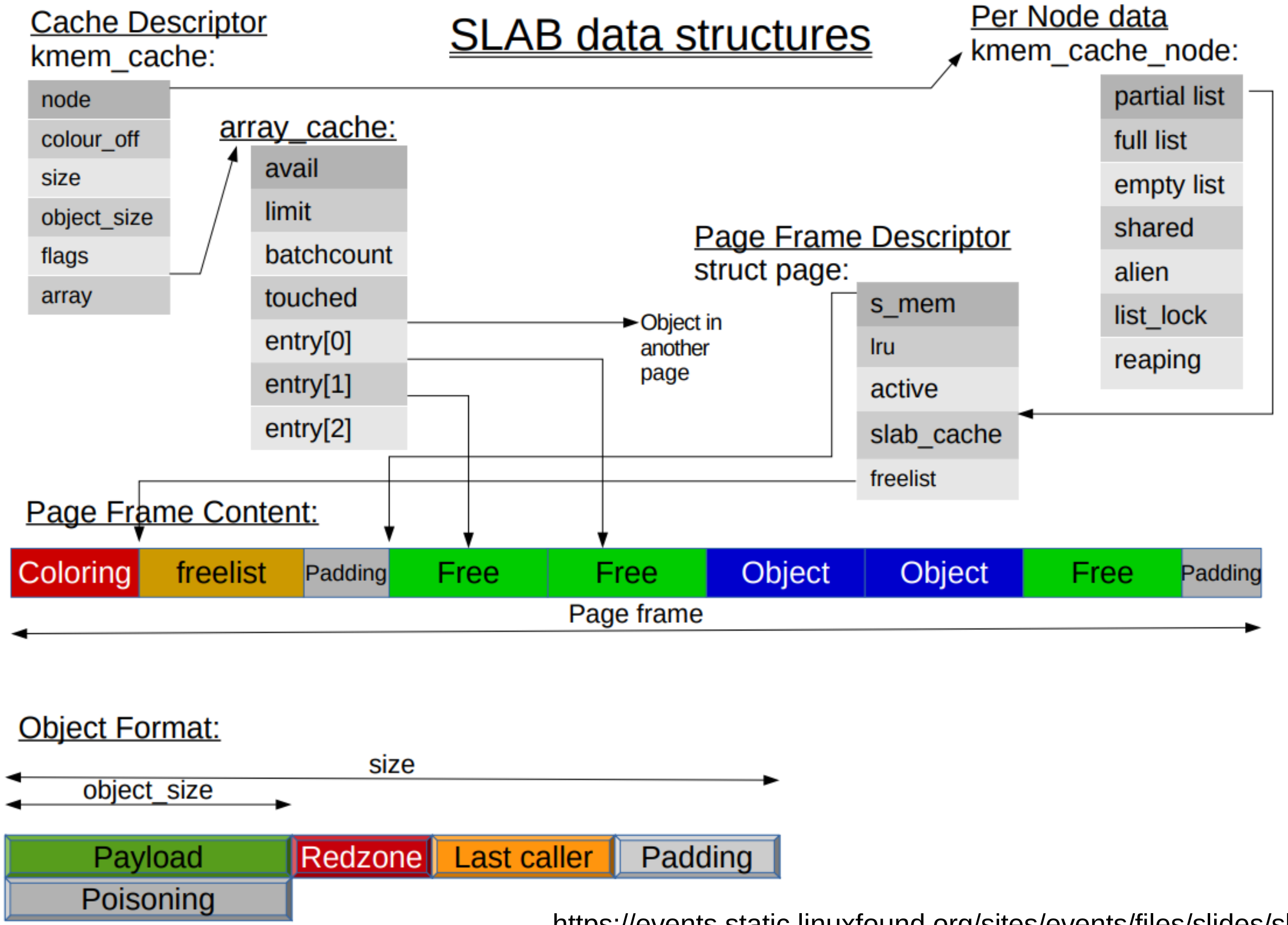


https://events.static.linuxfound.org/sites/events/files/slides/slaballocators.pdf

SLAB allocator – data structure

- **Red zone**
  - Used to detect writes after the object
- **Poisoning**
  - If the object is inactive then the bytes contain **poison** values
- **Padding**
  - An unused data to fill up the space to get the next object properly aligned
- **Coloring**
  - A scheme that attempts to <u>have objects in different slabs use different lines in the cache</u>
  - Objects use different cache lines ensure objects from the same slab cache will be unlikely to flush each other

SLAB data structures



https://events.static.linuxfound.org/sites/events/files/slides/slaballocators.pdf

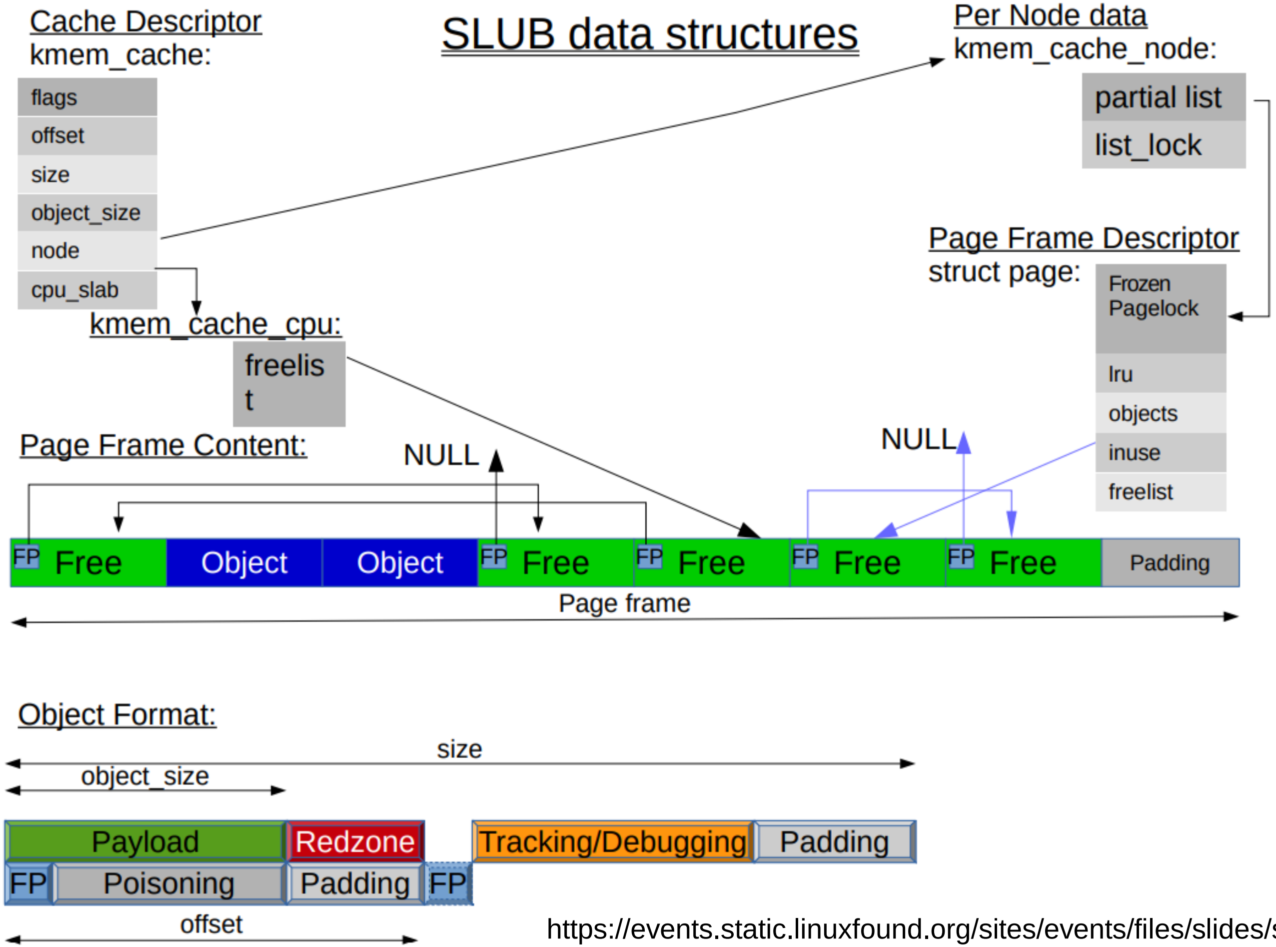
SLOB allocator

- **Small systems**
  - The bookkeeping overheads become critical on tiny memory system such as embedded systems
- **Simple list of blocks (SLOB)**
  - Just keep a free list of each available chunk and its size
  - Currently uses a first-fit algorithm
  - Grab the first one big enough to work
  - Split block if leftover bytes
  - No internal fragmentation
  - External fragmentation? Yes. Trade for low overheads

SLUB allocator

- **Large system**
  - The number of SLAB queues can make allocation fast but add complexity and storage overhead in large systems
- **The unqueue slab allocator (SLUB)**
  - All objects of same size from same slab
  - **Simple free list per slab – no per-slab metadata**
  - Add new fields in struct page to guide the search of free objects
    - void *freelist; // points to the first free object within a slab
    - short unsigned int inuse; // the number of objects allocated from the slab
    - short unsigned int offset; // tells the allocator where to find the pointer to the next free object

SLUB data structures



https://events.static.linuxfound.org/sites/events/files/slides/slaballocators.pdf

kmalloc allocator

- **kmalloc()**
  - Allocate memory for the kernel from general purpose caches
  - For small sizes, it relies on generic SLAB caches (see /proc/slabinfo)
  - For large sizes, it relies on the page allocator
  - The allocated area is guaranteed to be physical contiguous
  - The allocated area size is rounded up to the size of the smallest SLAB cache in which it can fit

kmalloc API

- #include <linux/slab.h>
- void ***kmaloc**(size_t size, int flags);
  - Allocate size bytes and return a pointer to the area (virtual address)
  - Size: number of bytes to allocate
  - Flags: same flags as the page allocator (GFP_KERNEL, GFP_ATOMIC, GFP_DMA, etc.)
- void **kfree**(const void *objp);
  - Free an allocated area

```
struct ib_port_attr *tprops;
tprops = kmalloc(sizeof *tprops,
GFP_KERNEL);
…
kfree(tprops);
```

drivers/infiniband/core/cache.c

vmalloc allocator

- The **vmalloc()** allocator
  - Used to obtain memory zones that are **contiguous in the virtual addressing space**, but not made out of physically contiguous pages
  - <u>The allocated area is in the kernel space part of the address space</u>
  - Allocations of fairly large areas is possible
  - Physical memory fragmentation is not an issue
  - Areas **cannot be used for DMA**, since DMA usually requires physically contiguous buffers
  - API in **include/linux/vmalloc.h**
  - void *vmalloc(unsigned long size); // return a virtual address

Conclusion

- Dynamic memory allocation
  - Fit for arbitrary complex data structure
- Buddy memory allocation
  - Simple, fast for power of two blocks
  - Fragmentation
- Slab memory allocator
  - Caching the commonly used objects

Takeaway Questions

- When does external fragmentation occur ?
  - (A) The size of each memory block is fixed
  - (B) The free space consists of variable-sized units
  - (C) The size of memory is large
- What are impacts of external fragmentation?
  - (A) No single contiguous space that can satisfy the request
  - (B) Require garbage collection to reclaim free memory space
  - (C) Increase the search time of the first fit allocation strategy

Takeaway Questions

- When are problem of the buddy allocator?
  - (A) Slow memory allocation
  - (B) External memory fragmentation
  - (C) Internal memory fragmentation
- What are problems of memory leaks?
  - (A) Run out of memory quickly
  - (B) Recycle storage while it's still being used
  - (C) Often occur in short-lived user processes

Takeaway Questions

- When are impacts of slab allocator?
  - (A) Caching frequent kernel objects
  - (B) Mitigate the external fragmentation
  - (C) The better utilization of hardware cache